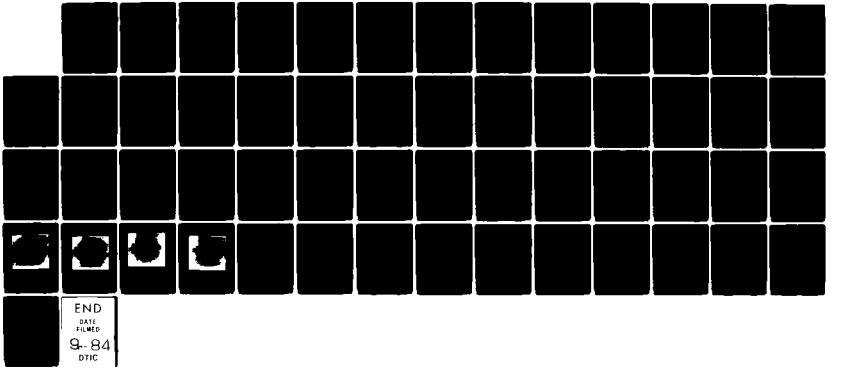
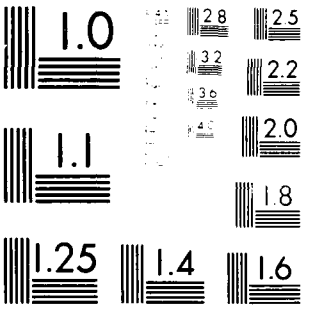
AD-A144 825 COGNITIVE PRINCIPLES IN THE DESIGN OF COMPUTER TUTORS 1/1
(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF
PSYCHOLOGY J R ANDERSON ET AL. 20 JUL 84 TR-84-1-ONR
UNCLASSIFIED N00014-84-K-0064 F/G 5/9 NL



END
DATE FILMED
9-84
DTIC



MICROCOPY RESOLUTION TEST CHART

Cognitive Principles in the Design of Computer Tutors

John R. Anderson

C. Franklin Boyle

Robert Farrell

Brian J. Reiser

Advanced Computer Tutoring Project

Carnegie-Mellon University

AD-A144 825

DTIC FILE COPY

Reproduction in whole or part is permitted for any purpose of the United States Government. This research was sponsored by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research under Contract No. N00014-84-K-0064, Contract Authority Identification Number NR 667-530. Approved for public release, distribution unlimited.

84 08

SECURITY CLASSIFICATION, THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: Technical Report #ONR-84-1	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Cognitive Principles in the Design of Computer Tutors	5. TYPE OF REPORT & PERIOD COVERED: Interim Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): John R. Anderson, C. Franklin Boyle, Robert Farrell, and Brian Reiser	8. CONTRACT OR GRANT NUMBER(s): N00014-84-K-0064
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 667-530
11. CONTROLLING OFFICE NAME AND ADDRESS: Personnel and Training Research Programs, Office of Naval Research, Arlington, VA 22217	12. REPORT DATE: July 20, 1984
	13. NUMBER OF PAGES: 51
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

artificial intelligence
cognitive science
tutoring
problem-solving
instruction
production systems
LISP
geometry

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A set of principles are derived from the ACT theory for designing intelligent tutors: identify the goal structure of the problem space, provide instruction on the problem-solving context, provide immediate feedback in errors, minimize working memory loads, use production system models of the student, adjust the grain size of instruction according to learning principles, enable the student to approach the target skills by successive approximation, and promote use of general problem-solving rules over analogy. These principles have successfully guided our design of tutors for LISP and geometry.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Reproduction in whole or part is permitted for any purpose of the United States Government. This research was sponsored by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract No. N00014-84-K-0064, Contract Authority Identification Number, NR 667-530. Approved for public release; distribution unlimited.

A-1

Abstract

A set of principles are derived from the ACT theory for designing intelligent tutors: identify the goal structure of the problem space, provide instruction on the problem-solving context, provide immediate feedback in errors, minimize working memory loads, use production system models of the student, adjust the grain size of instruction according to learning principles, enable the student to approach the target skills by successive approximation, and promote use of general problem-solving rules over analogy. These principles have successfully guided our design of tutors for LISP and geometry.

There has been and continues to be a great deal of hope for the role of computers in education (Cohen, 1982; Papert, 1980; Taylor, 1980). The actual record of accomplishment is still quite modest. Most computer education takes the form of drill and practice and is often not as effective as classroom drill and practice. There has always been the hope that artificial intelligence techniques would illuminate computer instruction. The buzz word of a decade ago was "intelligent computer assisted instruction" or ICAI (Carbonell, 1970). The relative lack of progress in that field led to a disenchantment with that term and we now see new words. The title that many now prefer, including ourselves, is intelligent tutoring (Sleeman and Brown, 1982).

The basic observation that motivates the intelligent tutoring approach is the great effectiveness of private human tutors over either classroom instruction or standard computer education. In comparisons of the human tutor with classroom instruction we have come up with estimates of well over 100% improvement in effectiveness (more will be said on this topic at the end of the paper). The hope of intelligent tutoring is to find some way of "bottling" the skill of the human tutor and putting it in a computer tutor.

Probably the expert systems methodology would be the most straight forward way of applying artificial intelligence to intelligent tutoring. Treat the human tutor as the expert whose knowledge has to be extracted and build an expert system to apply that methodology. Work such as that of Stevens, Collins, and Goldin (1979) on teaching topics such as rainfall seems to have this character. However, human tutoring does not have the characteristics of a domain that proves susceptible to the expert system methodology. It is not a relatively circumscribed knowledge domain, it is heavily weighted on natural language understanding, and human tutors show enormous differences in their tutoring style. Thus, it seems unlikely that there is a crystalized expertise to be captured and emulated.

It is probably for this reason that most approaches to intelligent tutoring have not tried to really emulate actual tutors (see the papers in Sleeman and Brown, 1982). Rather they have tried to identify principles of effective tutoring and design tutors that embody these principles. These tutors have often involved expert systems as submodules in the tutor. For instance, Brown, Burton, and DeKleer (1982) have an expert circuit analysis system as part of their SOPHIE tutor for troubleshooting

circuits. Such a system can reason correctly about the domain and the correct answer is, of course, an important piece of information in instruction.

We feel that the actual pedagogical design of the tutor has to be based on detailed cognitive models of how students solve problems and learn. Past tutors have been based on intuitions about these matters but not on cognitive theories. The state of art in cognitive science has reached the point where we now are able to produce theories capable of applications to intelligent tutoring. In this paper we would like to develop a set of principles for computer tutoring based on the ACT theory of cognition (Anderson, 1983). This theory has been developed at a sufficient level of detail that it is possible to produce simulations of the theory that actually solve problems at the level students solve problems and which learn from problem-solving much as students learn. At the core of the tutors we have developed are what we call ideal models which model how successful students solve problems in the domain. Such models are one step more constrained than the expert systems of most tutors. Not only must they be able to solve the problems in the domain, they must be able to solve them the way students do.

The major portion of this paper will be devoted to identifying and justifying some principles for doing intelligent tutoring. Then the last two sections of the paper will describe two tutors we have built partially embodying these principles. One is a tutor for LISP and the other is a tutor for high school geometry. However, before embarking on any of this we should state a major limitation on the principles that we will articulate. We can only defend their application to a relatively restricted set of topics--those for which we can develop ideal student models. These are the domains of high school and early college math and introductory programming. These domains are relatively sparse in their importation of extra-domain knowledge which is why they are relatively easy to develop ideal models for. We cannot tutor topics for which we lack precise student models in the same way as we can tutor the domains for which we do. If one does not know exactly what it is the student is supposed to do, it forces one to back off into a different tutoring strategy.

Throughout this paper we will be making reference to observations we have made of high school students learning geometry and college students learning to program in LISP. We have

observed four students spend approximately 30 hours studying beginning geometry and three students similarly spending 30 hours learning LISP. These sessions have been all recorded and have been subjected to varying degrees of analysis. Some of these analyses have been reported in a series of prior publications (Anderson, 1981; Anderson, 1982; Anderson, 1983a; Anderson, Farrell, and Sauers, 1984; Anderson, Pirolli, and Farrell, in press). This data base has served as a rich source of information about the acquisition of problem solving skill and has heavily influenced our design of computer tutors.

Principle 1: Identify the Goal Structure of the Problem Space

According to the ACT theory, and indeed most cognitive theories of problem solving, the problem-solving behavior is organized around a hierarchical representation of the current goals. It is important that this goal structure be communicated to the student and instruction be cast in terms of it. Below we discuss the fact that it is not communicated in typical instruction for LISP or geometry and some of the unfortunate consequences of this fact.

Geometry

Figure 1 shows the two-column proof form that is almost universally used in geometry. It is basically a linear structure of pairs where each pair is a statement and justification. Typical instruction encourages the belief that the goal structure of the student should mimic this linear structure -- that at any point in the proof the student will have generated an initial part of the structure and the current goal is to generate the next line of the structure.

Insert Figure 1 about here

There are two serious flaws with using linear proofs as goal structures. First this practice denies the validity of problem-solving search. It encourages the idea that the correct next line should be obvious, but finding the next line often involves considerable planning and search. Students engage in search but feel bad about themselves because they do. Second, search in such a linear structure like Figure 1 is doomed to be hopelessly unguided. If the only constraint is to generate a legal line, the search space for the correct proof is hopelessly large.

We have observed students fail at solving geometry problems because they try to work within this linear goal structure. In our limited data base (four students) we have found all students eventually overcome this linear conception and arrive at a better conception of the problem. However, our students had to go through an unnecessary amount of frustration before developing the necessary insights about the problem space.

Figure 2 illustrates the mental representation that we believe a successful student creates for the proof of a geometry problem. The conclusion to be proven is related to the givens of the problem by a hierarchical structure. In that structure rules of geometry relate givens to intermediate statements and these statements to the conclusion. Hopefully, the readers will find nothing surprising in this representation of the logical structure of the proof, but it needs to be emphasized that conventional instruction does not communicate this structure and students hardly find it obvious. This deficit is particularly grievous because we believe that the successful student's goal structure is much more closely related to this hierarchical proof structure than it is to the linear structure of a two-column proof.

Insert Figure 2 about here

Figure 3 illustrates the proof in Figure 2 embedded in a set of additional inferences generated by one of the authors (JRA) in constructing the proof. The inferences are numbered in the order that they were made. The structure includes many inferences that were made in an attempt to construct the proof but were not part of the final proof. These extra inferences reflect some of the search that was involved in producing the proof. Because standard pedagogy fails to communicate either the proof structure or the search based upon it, there is no way to provide instruction about this critical aspect of the learning process. This aspect is entirely a matter for the student to induce. Given its complexity, it is no wonder that students have the difficulties they do with geometry proofs.

Insert Figure 3 about here

LISP Programming

As in geometry, the goal structure underlying writing a LISP function does not correspond to the syntax of the problem solution and yet instruction is unfortunately cast in terms of the syntax. Our studies indicate that generating a LISP program is largely a top-down, planning process (Anderson, Farrell, and Sauers, 1982; Anderson, Pirolli, and Farrell, in press). The surface form of most programming languages involves a linear sequence of symbols. Thus, there is the natural danger of casting instruction in the terms of this linear structure. Fortunately, more enlightened instruction does emphasize a hierarchical, structured program. There is evidence that this is a better instructional mode (Shneiderman, 1980) although it is notoriously difficult to prove obviously correct hypotheses in this field with conventional experimental methodology (Sheil, 1981).

While structured programming is definitely a step in the right direction, it only ameliorates the basic problem. The structured program itself is only a syntactic object which will have an imperfect correspondence to the structure of the programmer's plan. Consider an example we have studied at length (Anderson, Pirolli, & Farrell, in press) from learning to program recursive functions, a recursive function that calculates the intersection of two lists:

```
(DEFUN INTERSECTION (LIST1 LIST2)
   (COND ((NULL LIST1) NIL)
         ((MEMBER (CAR LIST1) LIST2)
              (CONS (CAR LIST1)
                    (INTERSECTION (CDR LIST1) LIST2)))
         (T (INTERSECTION (CDR LIST1) LIST2))))
```

From a syntactic point of view, this function consists of a conditional structure that is composed of three clauses, each consisting of a condition and an action. The student is encouraged to believe that it is just a matter of "programmer's intuition" that leads to the division of the conditional into the right set of three conditions and actions.

However, there are very precise principles that underly the division of this code into its components (Soloway,1980; Rich & Shrobe, 1978). This is an example of what we call the CDR-recursion plan. This plan applies when an argument of the function is a list such as LIST1. That plan involves coding a terminating case for when the list is NIL and a recursive case that relates the result

of the function applied to the tail (CDR) of the list to the result of the function applied to the full list. In the code above, the first clause of the conditional implements the terminating case and the second and third clauses perform the recursive step. So, the plan that generated the conditional consists of two major subgoals, not three. However, it is necessary to break the recursive step into two subcases--one to deal with the situation where the first element is in the list and one where it is not. This division is dictated by a standard list search plan.

Experts code the function by implementing their CDR-recursion plan. However, this powerful programming technique is seldom directly taught. Students are given explanations of how the recursive calls work, but they are not told the programming plan that would cause someone to write these calls in the first place. We have seen students work for 10 hours with some popular LISP texts on problems like INTERSECTION and not figure out the CDR-recursion plan. As a consequence, they were still floundering after the 10 hours.

Another feature of problem-solving in LISP is that the programmer often has to interrupt the coding and go to a different problem space for algorithm design (Kant & Newell, 1982). As an example of algorithm design consider how the student *determines the recursive step in CDR-recursion*. A typical strategy is to list some examples of what the value of the function is for a set of example lists and for their tails and try to induce a general characterization of the relationship between the value of the function for the whole list and for the tail of the list. Inducing this characterization frequently involves pattern-matching. Listing examples and pattern-matching are not code generation activities but are critical to determining the code. Again standard instruction is silent about this algorithm design process.

Summary

Traditional instruction does not explain the goal structure of the problem-solving plan nor the search involved in the problem solving. Private human tutors often communicate this information in their real-time comments about the student's problem-solving. For instance, all three of the LISP tutors we observed suggested enumerating examples to discover the recursive relationship. A computer tutor should strive to communicate the goal structure to the student as this is absolutely

critical to effective problem solving.

Principle 2: Provide Instruction in the Problem-Solving Context

Students appear to learn information better if they are presented with that information during problem solving rather than during instruction apart from the the problem-solving context. There are a number of reasons why this should be so:

First, there is evidence that memories are associated to the features of the context in which they were learned. The probability of retrieving the memories is increased when the context of recall matches the context of study (Tulving, 1983; Tulving and Thomson, 1973). An extreme example of this was shown by Ross (1984) who found that secretaries were more likely to remember a text-editor command learned in the context of a recipe if they were currently editing another recipe.

Second, it is often difficult to properly encode and understand information presented outside of a problem context and so its applicability might not be recognized in a problem context. For instance, students may not realize that a top-level variable is really the same thing as a function argument even though they are obliquely told so. As another example, many students reading the side-angle-side *postulate may not know what included angle means and so misapply that postulate.*

Third, even if student can recall the information and apply it correctly, they are often faced with many potentially applicable pieces of information and do not know which one to use. We have frequently observed students painfully trying dozens of theorems and postulates in geometry before finding the right one. The basic problem is that knowledge is taught in the abstract and the student must learn the goals to which that knowledge is applicable. If the knowledge is presented in a problem-solving context its goal-relevance is much more apparent.

Private human tutors characteristically provide information in the problem-solving context. Some, but not all, of the tutors we observed gave almost non-stop comment as students tried to solve problems. They take great advantage of the multi-modality character of the learning situation--with the student solving the problem in the visual modality and their instruction in the auditory modality. Although it would be difficult for a computer tutor to be as interactive as these human tutors and take full advantage of multi-modal processing, we shall see that it is possible to partially achieve this by

providing appropriate instruction tailored to the students' current goal context.

Principle 3: Provide Immediate Feedback on Errors

Novices make errors both in selecting wrong solution paths and in incorrectly applying the rules of the domain. Errors are an inevitable part of learning, but the cost of these errors to the learner is often higher than is necessary. They can severely add to the amount of time required for learning. More than half of our subjects' problem-solving sessions were actually spent exploring wrong paths or recovering from erroneous steps. Relatively little is learned while students are trying to get out of the holes they have dug for themselves.

In addition, errors often confuse the picture and make it difficult to determine which steps were right or wrong. The classic example of this is the student who finally stumbles on the correct code but does not understand why it works. Students often progress in this trial and error mode with respect to LISP evaluation: they don't know when an element will be treated as a function, a variable, or a literal but play around with parentheses and quotes until they get something to work. It is particularly difficult to learn from errors when the feedback on the errors comes at a delay. We (Lewis & Anderson, submitted) have shown that subjects learn more slowly in a problem-solving situation where they are allowed to go down erroneous paths and are only given feedback at delay. Also, the importance of immediacy of feedback has been well documented in other learning situations (Bilodeau, 1969; Skinner, 1958).

Another cost of errors is the demoralization of the student. These are domains in which errors can be very frequent and frustrating. We believe that much of the negative attitudes and math phobias derive from the bitter experiences of students with errors.

The advantages of a private tutor are clear in this regard. The tutor can prevent the student from wasting inordinate amounts of time searching wrong paths. The tutor can both provide immediate feedback when errors are made and point out to the students which aspects of the problem-solution are correct and which are in error.

Principle 4: Minimize Working Memory Load

Solving problems can require holding all the requisite information in a mental working memory. If some of that requisite information is lost there will be errors. It surprised us to find out in our LISP protocols that most of the student errors appear to be due to working memory failures. A frequent and disastrous type of error is "losing a level of complexity". One way this manifests itself is that subjects lose track of one level of parentheses. Another way this occurs is when subjects plan to use function1 within function2 within function3, but forget the intermediate function and write function3 directly within function1.

Working memory errors are frequent in geometry but less frequent than in LISP. (Their lesser frequency is because the diagram and the proof are effective records of inferences made.) One place that working memory errors do occur is when students work backward from the goal and create subgoals. For instance, a subject will reason "I can prove these segments congruent if I can show this triangle is a right triangle. I can do that if I can show its other two angles sum to 90". The subject then proves the two angles sum to 90 but forgets how this fits into the bigger picture. It is because of the high working memory burden that students are so bad at backward inference in geometry.

A good human tutor can recognize errors of working memory and typically provides quick correction (McKendree, Reiser, and Anderson, 1984). Tutors realize that there is little profit in allowing the student to continue with such errors. However, human tutors really have no easy means at their disposal for reducing the working memory load. This is one of the ways we think computer tutors can be an improvement over human tutors--one can externalize much of working memory by rapid updates in the computer screen. This involves keeping partial products and goal structures available in windows.

Principle 5: Represent the Student as a Production Set

All of our work on skill acquisition has modeled students' behavior as being generated by a set of productions. There is a fair amount of evidence for this view of human problem-solving (e.g., Anderson, 1983; Newell & Simon, 1972). It is also the case that numerous other efforts in the domain of intelligent tutoring have taken to representing the to-be-tutored skill as a production set (eg. Brown

and Van Lehn, 1980; O'Shea, 1979; Sleeman, 1982).

Productions in ACT represent the knowledge underlying a problem-solving skill as a set of goal-oriented rules. Some representative examples for LISP and geometry are:

```
IF      the goal is to insert a element into a list
THEN    plan to use CONS and set as subgoals
                  1. To code the element
                  2. To code the list

IF      the goal is to code a function that calculates a relation on a
        list
THEN    try to use CDR-recursion and set as subgoals
                  1. To code the terminating condition
                  2. To code the recursive condition

IF      the goal is to prove △XYZ ≅ △UVW
                  and XY ≅ UV
                  and YZ ≅ VW
THEN    plan to use side-angle-side and set as a subgoal
                  1. To prove ∠XYZ ≅ ∠UVW
```

Such rules not only enable the system to follow student problem-solving but they define an appropriate grain size for instruction. Basically, the tutoring systems monitor whether a student has each rule correctly and corrects any incorrect or missing rules. As emphasized by Brown and Van Lehn, student misconceptions or bugs can be organized as perturbations of such rules.

Human tutors try to intuit an appropriate grain size of rules for instruction but often their intuitions are wrong. This is one place where a system based on careful analysis of student problem-solving may be able to outperform the typical human tutor.

Principle 6: Adjust the Grain Size of Instruction with Learning

One of the reasons human tutors have difficulty with the grain size for instructing students is that the grain size changes as experience is acquired in the domain. According to the ACT learning theory this change is produced by a knowledge compilation process that collapses sequence of productions into larger "macro" production rules. Human tutors, being highly skilled in the domain exemplify a large grain size in their problem-solving and have a considerable difficulty intuiting the appropriate grain size for the student.

An effective computer tutor will have to adjust the grain size of instruction as the student progresses through the material. Using a theory of production learning, it will have to predict when

the original productions become compiled into macro productions so that it can change the grain size of instruction.

Principle 7: Enable the Student to Approach the Target Skill by Successive Approximation

Students do not become experts in geometry or LISP programming after solving their first problem. They gradually approximate the expert behavior, accumulating separately the various pieces (production rules) of the skill. It is important that a tutor support this learning by approximation. It is very hard to learn in a tutorial situation that requires that the whole solution be correct. The tutor must accept partially correct solutions and shape the student on those aspects of the solution that are weak.

Generally, it is better to have the early approximations occur in problem contexts that are as similar to the final problem space as possible. Skills learned in one problem context will only partially transfer to a second context. Students are learning features from early problems to guide their problem-solving operators. If these features are different from the final problem space the problem-solving operators will be misguided. For instance, early problems in geometry tend to involve algebraic manipulations of measures. Consequently, the student learns to always convert congruence of segments and angles into equality of the measure of the parts. Later problems such as those involving triangle congruence do not involve converting congruence of sides and angles into equality of measures. Students frequently carry over their overgeneral tendency to convert and get into difficulties because of this.

It is often extremely difficult for students starting out to solve the kind of problems that they will eventually have to solve. For instance, in geometry students cannot initially generate proofs. To deal with this, standard pedagogy often evolves intermediate tasks such as giving reasons for the worked out steps of a proof. The problem is that the process of finding reasons for the steps of a proof is different than the process of generating that proof. As another example, in programming students are often given practice evaluating recursive functions as preparation for writing recursive functions. Again, these are separate tasks. Both in the geometry and the programming case we have shown that

there is not much transfer from one task to another.

The advantage of a private tutor is that he/she can help the student through problems which are too difficult for the student to solve entirely alone. Thus, it is common to see a sequence of problems where the tutor will solve most of the first problem with the student just filling in a few of the steps, less of the second, etc. until the student is solving the entire problem.

Principle 8: General Problem-Solving Rules over Analogy

There are two basic methods that we have observed students using to solve problems in these domains. One is to use analogies to earlier problems in the text or problems from other domains to help guide the problem solving. The basic strategy is to try to map the structure of a solution of one problem to another problem. Anderson (1981 tech report), Anderson, Farrell, and Sauers (1984), and Anderson, Pirolli, and Farrell (in press) discuss specific examples from our protocols on geometry and LISP.

The other method is to extract general problem-solving operators from the instruction and apply these to the problem. For instance, if the goal is to prove triangles congruent, one can apply postulates about triangle congruence. If the goal is to create a list structure one can try to apply a function that creates list structures. The problem with such general operators is that in many domains the search space of the combinations of these operators becomes enormous. This is perhaps why only little additional information tends to be introduced with each new section of a textbook. The student can restrict search to these new potential operations (cf. Van Lehn, 1983).

Another difficulty with the general problem-solving approach is that it is often difficult to encode the needed problem-solving operators. Often the instruction does not contain explicit statement of such operators. Rather the operators have to inferred from the instruction. Even on those occasions in which the operators are directly stated, students have a hard time understanding them because they are stated so abstractly. Students are often only able to encode the operators correctly when they see them applied to an example problem.

Subjects appear to prefer analogy as a method of solution in both geometry and LISP. The preference is not overwhelming in geometry and there are many episodes of problem solution by

general problem-solving operators. In contrast, the preference is overwhelming in novice LISP programming. In almost every case where a student was writing a first instance of a particular type of LISP function, the student relied on analogy to example LISP functions.

Private human tutors differ as to whether they tend to guide the student to solution by analogy or by general problem-solving operators. We claim that solution with general operators would lead to the best long-term gains. This is because the student often successfully generates a solution by analogy but does not understand why the solution works. We have seen students work their way through problems by analogy and not learn anything of permanent value. What they often learn is how to do analogies. Take away the problems from which to analogize and they are unable to solve problems. Halasz and Moran (1982) have also commented on the negative consequences of problem solving by analogy. They point out that students are prone to incorrect inferences in using the analogy. An analogy is frequently used to replace a deep understanding of the problem domain. Learning is going to be limited if students do not really understand the domain.

Given the importance of analogy to current thinking to cognitive science, the issue of analogy versus general problem-solving needs further research. However, we have structured our tutors to get subjects to solve the problems with general problem-solving operators. We prevent the student from having access to previous solution. Rather when students need help we provide them with general descriptions of the relevant problem-solving operators.

In saying that we discourage problem-solving by analogy we are not saying that examples are not important to learning. As discussed early it is much easier to understand the problem-solving operators when they are presented in the context of an example. However, we try to keep just one example in front of the student, the current problem being worked on, and not give the subject access to prior worked-out examples.

Issues of Human Engineering

We have tried to achieve the principles enumerated above in our development of computer tutors. However, many of the problems that we face in creating actual computer tutors were not with achieving these principles but were at a level which might best be called human-engineering. This

refers to issues of designing the interface and the natural language dialogue so that information exchanges occur that satisfy our cognitive principles. Our human engineering efforts have been somewhat guided by what we know from cognitive psychology, somewhat guided by results in the literature on the human-computer interface, but largely the result of trial and error exploration. This human-engineering aspect is far from trivial. Before we are going to have a good theory of intelligent tutors, we will need a good theory of their human engineering.

In the remaining two sections of this paper we will describe our geometry tutor and our LISP tutor. We will try to show how these tutors approximate the design criteria we have set forth. We will also will try to communicate some of our experience with the human-engineering problems.

The Geometry Tutor

A geometry proof problem as stated in a high-school geometry text (see Fig. 1) consists of three ingredients--a diagram, a set of givens, and a to-be-proven statement. Despite frequent claims to the contrary, the diagram frequently plays a critical role in the logical proof. Typically the diagram of the only source of critical information about what points are collinear and which points are between which others. This information often is not provided in the givens. Most other information which can be read off the diagram, such as relative measure, is not to be taken as true in general. It is a rare high-school proof problem that involves constructions or creating new entities not in the diagram. Indeed, some geometry textbooks have a policy of never requiring the student to do proofs by constructions although all conventional texts must use proof by construction in establishing theorems for the student. Therefore, to a good approximation, the student's task is to find some chain of legal inferences from the stated givens and the givens implicit in the diagram to the conclusion.

Insert Figure 4 about here

The Ideal Model

Consider the problem in Figure 4. There are a set of forward and backward deductions that can be made. Forward deductions take information given and note that certain conclusions follow. So,

we can infer from the fact that the M is the midpoint of $\overline{AB}$ that $\overline{AM} \cong \overline{MB}$. We can also infer from the vertical angle configuration that $\angle AMF \cong \angle BME$. Backward inferences involve noting that a conclusion could be proven if certain other statements were proven. So, we could prove M is the midpoint of $\overline{EF}$ if we could prove $\overline{EM} \cong \overline{MF}$. We could prove the latter statement if we could prove $\overline{EM} \cong \overline{AM}$ and $\overline{MF} \cong \overline{AM}$. As these examples illustrate, sometimes forward and backward inferences are part of the solution and sometimes they are not. In challenging problems the student cannot always know whether an inference is part of the solution. Rather, the student can make heuristic guesses at which inferences are likely.

In our view, the ideal student extends the proof backward from the to-be-proved statement and forward from the givens until there is a complete proof. At each point the ideal student makes the heuristically best inference where this is defined as the inference most probably part of the final proof. "Most probable" depends on some induction over the space of high-school problems. Currently, we have no formal definition of the probability that an inference will part of a proof, but our intuitions are usually quite defensible. So, we create the ideal student model as a set of rules that seem heuristic. For instance, one rule is that when vertical angles are parts of to-be-proven congruent triangles, infer that they are congruent but not otherwise. Basically, the rules all take the form of "apply a particular rule of inference when such and such conditions prevail". These conditions refer to properties of the diagram, givens, and established inferences, and goals set in backward inference. These rules can be represented as production rules; for example:

IF	$\triangle XYZ$ and $\triangle UVW$ are to-be-proven congruent triangles and $\overline{XYW}$ and $\overline{ZYU}$ are intersecting lines
Then	infer $\angle XYZ \cong \angle UYW$ because of vertical angles.

The Proof Graph

As noted earlier, standard instruction does a very poor job of communicating the goal structure to the student. Therefore, our first human-engineering problem was to find a way of communicating this information to the student. We decided to use a graphical formalism in which the to-be-proven statement was at the top and the givens were at the bottom. A proof is created as a logical network connecting the givens to the to-be-proven statement. The basic unit of this network is a structure

connecting one or more givens to a conclusion through a rule of inference. Figure 5 shows four states of the proof network from beginning to end for the problem in Figure 4. The network can be grown from the bottom by the forward inference or from the top by backward inference. As Figure 5 illustrates, it is certainly possible to generate inferences off the correct path. We have testimonials from two pilot students that they better understood both the structure of the proof and nature of our proof generation because of the graphical formalisms of the proof structure.

Insert Figure 5 about here

Interacting with the System

The basic cycle of interaction with the system is as follows: The student points to one or more statements on the screen from which he or she wants to draw an inference. If at least one legal inference can apply to these the statements will blink and the system asks the student for the rule of inference to apply. The student types in the rule of inference. If it is applicable to these statements the system will prompt the student for the conclusion that follows. The student types in the conclusion or points to it if it is on the screen. If correct the student can now initiate another cycle of interaction with the system. This continue until a complete proof has been generated.

The human-engineering of the system has involved a lot of trial and error to decide issues such as how to position the graph, what abbreviations to use, when to correct misspellings, how to let the student point, and how to relate the proof structures to the diagram. We have also found it useful to have the student spend an hour with a warm-up system that uses the same graphical conventions but with the familiar domain of arithmetic. The student can learn to use the system much more rapidly if this learning is decoupled from learning to do proofs in geometry. One problem with a complex screen is that the student may not notice when new information is added. We have found that color and motion are fairly effective ways of capturing attention. So we have taken to changing the colors of the windows, bringing up new windows in new color, or blinking critical information.

For the accomplished geometry student, this system is a convenient and efficient vehicle for constructing proofs. It serves as an external memory so that forgotten information can be quickly

recovered. It also catches slips of mind quickly. However, more is needed when dealing with a novice. The most basic problem for a student is not knowing how to proceed. There are a number of ways that our system helps students during problem solving. Most directly the student can ask the system for help. Less directly, the student may choose to make inferences off nodes from which no inferences or no useful inferences follow. In either case, the system will provide the student with a hint in the form of suggesting the best nodes from which to infer.

The student may be uncertain about what inference rule to apply or how to apply the inference rule. He can manifest this again by asking for help or by inappropriately applying a rule. In this case pop-up windows can be brought up to display which rules of inference are currently applicable and to display a definition of each rule of inference. When the student displays a known bug, such as applying the side-angle-side postulate when the angles are not included by the sides, a pop-up window will appear explaining why the student's choice is not correct.

Students get in ruts in which they try to make an inference from an inappropriate set of statements over and over again or apply the same rule over and over again. In such cases, the system will interrupt and display what it regards as the next best inference step and interrogate the student to make sure that the step is understood.

An interesting property of the graphics screen is that the student has no access to solutions of previous problems. Therefore, it is very difficult to do any problem solution by analogy. When the student gets instruction, the instruction is about the inference rules in the form of general problem-solving operators. For instance, the system will give the following statement of the corresponding parts rule when it is evoked in backward inference to prove two sides congruent:

IF you want to prove the conclusion $\overline{UV} \cong \overline{XY}$

THEN try to prove the premise $\triangle UVW \cong \triangle XYZ$

along with a pattern diagram to help the student instantiate the abstract terms in this statement.

We have looked intensively at three students working with the system--one with above average ability, one of average ability, and the other of below-average ability (as defined by their math grades). The below-average student came to us for remedial purposes having failed 10th grade geometry. The

other two students were eighth graders with no formal geometry experience. We can only report that the system works--that all students learned with it and without great difficulty. We think they learned faster than with traditional instruction, but we have no way to document this belief. All students were able to do problems that local teachers consider too difficult to assign to their 10th grade classes. We are currently studying additional students working with a more advanced version of the geometry system.

The LISP Tutor

Our work on the LISP tutor is based on earlier research (Anderson, Farrell, Sauers, 1984; Anderson, Pirolli, and Farrell, in press) studying the acquisition of basic LISP programming skills by programming novices. Given standard classroom instruction, a programming novice typically takes over 50 hours to acquire a basic facility with the data structures and functions of LISP. At the end of this period they can write basic recursive and iterative functions. In this time the student has probably not written a LISP program more than three functions deep and still does not know how to use LISP for interesting applications. It is at least another 50 hours before the student has the facility to write modest problem-solving programs.

We believe that the LISP tutor will be able to cover the same material in under 20 hours (we are only a third of the way there so far). After this point the tutor would step back and become an "intelligent editor" which could help the student create programs and catch obvious slips, but would no longer instruct. Besides the desire to have a manageable project, we do not feel we are capable of modeling the problem-solving that occurs after learning the basics of LISP.

The Ideal Model

Brooks (1977) analyzed programming into the activities of algorithm design, coding, and debugging. We are currently focusing on algorithm design and coding. Our past research on LISP involved creating simulations of both errorful and ideal students in the algorithm design and coding phases. Brooks characterized programming as first designing an algorithm and then converting it into code. However, the break is seldom so clean. The student alternates between algorithm design and coding, sometimes omitting the algorithm design altogether and going directly from problem

statement to code. Often novices and experts differ as to whether there is a distinct algorithm design stage. One example we have studied involves writing LISP code to take a list and return that list with the last element removed. A number of experts generated (REVERSE (CDR (REVERSE LIST))) immediately upon hearing the statement whereas some novices went through a 10 minute phase of means-ends analysis to come up with the algorithm.

The ideal model for code generation, both for experts and novices, involves a top down generation of the code. Figure 6 illustrates the goal structure underlying the generation of the code for the function POWERSET which takes a list and returns the list of all sublists. This is recognized as involving recursion on the list and subgoals are set to code the terminating step and the recursive step of the recursion Both of these steps are broken down into algorithm design plus code generation. Writing the code for the recursive step involves writing a "helping function" ADDTO that will add an element to each list in a list of lists. Novices and experts differ in whether they will try to write the function ADDTO before completing the code for POWERSET. Novices tend to be uncertain exactly what ADDTO will do and whether they can actually write such a function. Therefore, they will interrupt their writing of POWERSET to code ADDTO. Experts are confident that ADDTO can be coded and postpone the writing of ADDTO until completing POWERSET.

Insert Figure 6 about here

The actual code generated along with the goal structure in Figure 6 is:

```
(defun powerset (list)
      (cond ((null list) (list nil))
            (t (append (powerset (cdr list))
                       (addto (car list) (powerset (cdr list))))
             )))
```

There are a number of features to note about Figure 6. First, code is generated top-down. Second, there is alteration between algorithm design and code generation. Third, the encoding of the embedded function, ADDTO, is postponed until the top function is encoded.

It is difficult to develop ideal models for the algorithm design aspect of the problem-solving. Students can potentially bring any of their past experiences and prior knowledge to bear in designing an algorithm. With each problem we have to specially provide the ideal model with potentially

relevant prior knowledge. So, for instance, to model the creation of a graph search function we had to provide the system with knowledge of the fact that paths in a graph can loop (Anderson, Farrell, and Sauers, 1982).

Interacting with the LISP tutor

Figure 7 illustrates a typical state of the screen in interacting with the tutor. We used the hierarchical structure of LISP to support representation of the top-down structure of the programming activity. At any point in time the partial code is displayed before the student in one window with special markers indicating the structure requiring top-down expansion: eg

(defun rotator (lis) (append (last <1>) <2>))

where the symbols <1> and <2> denote the points for top-down expansion. The student selects a symbol for expansion and types code to replace this symbol. So, at this level the system is just a structured editor for creating the code.

Insert Figure 7 about here

Sometimes, the hierarchical structure of the LISP code does not correspond to the goal structure. A good example of this was the intersection function discussed earlier where the COND structure had three clauses but the goal structure for CDR-recursion just involved one goal for doing the terminating case and one goal for doing the recursive step. In such cases we have the subject generate the code according to the underlying goal structure rather than the syntax of the LISP code.

Sometimes, the student has to branch into algorithm design where the problem-solving will not correspond to the hierarchical structure of the code. In this case a window appears to support the development of the algorithm. Figure 8 illustrates a state of the tutor when the the algorithm window is being used to instruct the student. The final product of the algorithm designing will be a specific window of code that must be mapped into the abstract code of the function. So, for instance, if the student needs to design an algorithm for returning a list of all but the last of the LIS, we will have him work out a solution for a concrete instance of LIS, say LIS = (A B C D). The final product in this case would be:

(REVERSE (CDR (REVERSE '(A B C D)))

which can be mapped into the abstract function to produce:

(REVERSE (CDR (REVERSE LIS)))

Again, when doing the recursive step for POWERSET (see Figure 6) with the specific list (A B C), the final result in the algorithm window would be:

(POWERSET '(A B C)) =
(APPEND '(() (C) (B) (B C)) '((A) (A C) (A B) (A B C)))
and
(POWERSET '(B C)) = (() (C) (B) (B C))

We feel that the code window and the algorithm window do a good job of communicating the hierarchal structure of the programming activity as well as the separate status of algorithm design.

Insert Figure 8 about here

There is a third window in which the student communicates to the tutor his choices about code and algorithm. Processing the code is relatively straightforward but processing the algorithm is much more difficult. The most obvious method is to have the student type in an English description of his algorithm and for our tutor to try to understand that. The program has only the task of categorizing the student's description into one of the algorithm categories it is prepared to process. Even with this considerable constraint, we have only had modest success at language comprehension. One of the frequent student complaints about the tutor is its inability to understand algorithm descriptions. In part this is due to the tutor's limited natural language understanding and in part this is because students often only have vague ideas of algorithm choices. Our solution to both difficulties has been to implement a menu system.

Our menus are generated dynamically from the instantiations of productions in the ideal model. The menus contain descriptions of all the algorithmic variations, correct and buggy, that we are prepared to process. Menu selection is technically simpler than language comprehension. It is an issue for further research which is more effective. We were surprised by how well students appear to adapt to menu selection. This may be simply that it is much easier to pick a menu entry than to generate a description.

As in the geometry system, there are general help facilities so that the student can bring up

information. Again, the information is given in terms of abstract problem-solving operators. For example, rather than showing how CONS is used in a specific program, CONS might be explained: "If you want to insert an element into a list use the function CONS" with an example to instantiate the explanation.

Novices are prone to a substantial set of misconceptions and slips in writing LISP programs. One of the strengths of the system is that we have created recognizers for a great many of the possible bugs and provided for appropriate feedback on the errors. In our pilot study, we have discovered a number of other stereotypic errors which we have also entered into the system with appropriate feedback.

Table 1 illustrates a dialogue with our tutor, linearized rather than developed as a sequence of screen images. This interaction demonstrates how the tutor recognizes misconceptions, provides immediate feedback, and prevents students from extensively exploring incorrect paths.

Because of the availability of the college population, we have been able to evaluate the quality of our LISP tutor more carefully than we have the geometry tutor. We have a fairly robust version of the tutor that is able to take the student through basic LISP functions and data structures, evaluation, function composition, and function definition. It culminates in a series of six exercises in function definition. We estimate that students would take on the order of 10 hours in a standard classroom situation to listen to the lectures, read the text, and do the exercises. In contrast to that situation, students with our tutor took only an average of 1 hr, 37 min.

We took the written material associated with the computer tutor and had human tutors work personally with students. These human tutors were undergraduates and graduates who knew LISP well and who had previous experience tutoring. In this situation, they took 1 hr, 21 min. to work through the material. As a final condition we created a minimal instructional set that contained basically just the information in the computer tutor. Students were given this material and asked to work through the same problems as in the other condition. There was a consultant available should they get stuck. In this case subjects took 2 hrs 15 minutes to get through the material. So clearly either tutoring situation improved upon the on-your-own condition. The fact that the on-your-own

condition is better than the classroom probably reflects the fact that than standard textbooks or lectures waste a lot of time on irrelevancies.

Afterwards, subjects in all three conditions were administered a standardized test of knowledge of LISP. As a frame of reference, students after an estimated 30.5 hours of work in JRA's LISP class got 57% correct. Students with the computer tutor got 66% correct. Students with the human tutor got 69% correct. Students left on their own got 72% correct. There is effectively no difference in the performance of any of the three experimental groups of subjects, although they all seem to be doing better than the classroom subjects.

So, a private tutor, computer or human, is a considerable improvement over not having a private tutor. Humans are at least somewhat better than our computer tutor, but we were looking at a first pass version of the computer tutor. This tutoring experiment produced a set of data about how to do tutoring better for example, unexpected bugs and misconceptions. We really think it is feasible to expect that one can construct computer tutors that are better than the human tutors typically found in an university environment.

Final Conclusions

This is very much a report of work in progress. We have yet to get our tutors for geometry or LISP into their final states. We have yet to completely formalize our general methodology for creating tutors. Our evaluation is still very preliminary, both of the general methodology and of the specific tutors. However, we believe that the results are sufficiently encouraging to deserve a report now. The basic result is that it does seem possible to create computer tutors that are capable of making a major improvement in the education of a wide variety of topics. An important additional observation is that the computer technology to deliver these tutors is rapidly becoming economically feasible.

There are two important qualifications to make about any projections about the feasibility of creating intelligent tutors. First, design of these tutors depends critically on cognitive science expertise. It is not yet possible for a cognitively uninformed system designer to create an effective tutor nor for a computationally-uninformed educational psychologist to create such a system. We have tried to distill our research and design experience into a set of principles, but these principles

are incomplete: they do not specify how to create student models, and they do not specify how to engineer the human-computer interface. Therefore, creating a tutor for a new domain poses research problems that are still state of the art for cognitive science.

Second, developing even crude prototype systems is labor intensive and requires rather specialized laborers at that. We expect that we will spend 10 man-years of effort before we have satisfactory tutors for full courses in geometry or LISP. Even after that it will require considerable effort before these would be reliable and fine-tuned enough to be offered as commercial products. We hope to develop our methodology enough so that we can cut the 10 year figure at least in half. However, it seems unlikely that we will ever avoid the need for an intensive study of the domain and the behavior of student problem solvers .

Figure Captions

Figure 1. An example of a two column proof in high-school geometry.

Figure 2. A representation of the logical structure of inferential support.

Figure 3. A representation of the order of inferences made in discovering the proof in Figure 3.

Figure 4. A relatively advanced proof problem for high school geometry.

Figure 5. Four states of the tutor screen in a students construction of a graph proof for the problem in Figure 5.

Figure 6. A representation of the hierarchical goal structure underlying generation of the LISP function to calculate POWERSET.

Figure 7. A typical state of the tutor screen in interacting with a student

Figure 8. A state of the tutor screen during the tutoring of the algorithm. Note the middle window with a representation of the current state of the algorithm as apllied to a concrete example.

References

Anderson, J.R. (1981). Tuning of search of the problem space for geometry proofs. In **Proceedings of IJCAI-81** (pp. 165-170).

Anderson, J.R. (1981). **Acquisition of Cognitive Skill.** ONR Technical Report 81-1 Carnegie-Mellon University, Pittsburgh, PA.

Anderson, J.R. (1983). **The Architecture of Cognition.** Cambridge, MA: Harvard University Press.

Anderson, J.R., Farrell, R., & Sauers, R. (1982). **Learning to Plan in LISP.** ONR Technical Report ONR-82-2, Carnegie-Mellon University.

Anderson, J.R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP **Cognitive Science**, , in press.

Anderson, J.R., Pirolli, P. & Farrell, R. Learning recursive programming. In forthcoming book edited by Chi, Farr, & Glaser

Bilodeau, I. McD. (1969) Information feedback. In E.A. Bilodeau (Ed.), **Principles of Skill Acquisition** New York: Academic Press.

Brooks, R.E. (1977). Towards a theory of the cognitive processes in computer programming. **International Journal of Man-Machine Studies**, 9 737-751

Brown, J.S. & Van Lehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. **Cognitive Science** 4 379-426

Brown, J.S., Burton, R.R., & DeKleer, J. (1982). Pedagogical, naturla language and knowledge engineering techniques in SOPHIE I, II and III. In Sleeman & Brown (Ed.) **Intelligent Tutoring Systems** (pp 227-282). New York: Academic Press.

Carbonell, J.R. (1970). AI in CAI: An artificial intelligence approach to computer-aided instruction. **IEEE Transactions on Man-Machine Systems** 11 190-202

Cohen, V.B. (April 20, 1982). Computer software found weak **New York Times** C4 Summary of a research.

Halasz, F. & Moran, T.P. (March 15-17, 1982). **Analogy considered harmful.** Technical Report, Proceedings of the Human Factors in Computer Systems Conference, Gaithersburg, MD.

Kant, E. & Newell, A. Problem solving techniques for the design of algorithms. Proceeding of the Symposium on the Empirical Foundations of Information and Software Science, Atlanta, GA.

Lewis, M. & Anderson, J.R. The role of feedback in discriminating problem-solving operators.

McKendree, J., Reiser, B.J., & Anderson, J.R. Tutorial goals and strategies in the instruction of programming skills. Paper submitted to the 1984 conference of the Cognitive Science Society.

Newell, A. & Simon, H. (1972). **Human Problem Solving.** Englewood Cliffs, N.J.: Prentice-Hall.

O'Shea, T. (January, 1979). A Self-Proving Quadratic Tutor. **International Journal of Man-Machine Studies**, 11(1), 97-124.

Papert, S. (1980). **Mindstorms.** New York: Basic Books.

Rich, C. & Shrobe, H. (1978). Initial report of a LISP programmers' apprentice. **IEEE Trans. Soft. Eng.**, , 456-466.

Ross, B.H. (1984). Remindings and their effects in learning a cognitive skill. **Cognitive Psychology**, , . in press.

Sheil, B.A. (1981). The psychological study of programming. **Computing Surveys**, 13, 101-120.

Shneiderman, B. (1980). **Software Psychology.** Cambridge, MA: Winthrop.

Skinner, B.F. (1958). Teaching machines. **Science**, 128, 889-977

Sleeman, D. (1982). Assessing aspects of competence in basic algebra. In D. Sleeman & J.S. Brown (Eds.), **Intelligent Tutoring Systems** New York: Academic Press.

Sleeman, D. & Brown, J.S. (Eds.). (1982). **Intelligent Tutoring Systems.** New York: Academic Press.

Soloway, E.M. From problems to programs via plans: The context and structure of knowledge for introductory LISP programming. COINS Technical Report 80-19. University of Massachusetts at Amherst, 1980.

Stevens, A., Collins, A., & Goldin, S.E. (1979). **Misconceptions in student's understanding.**

Taylor, R. (1980). **The Computer in the School: Tutor, Tool, Tutee.** New York. Teachers College Press.

Tulving, E. (1983). **Elements of Episodic Memory.** London: Oxford University Press.

Tulving, E. & Thomson, P.M. (1973). Encoding specificity and retrieval processes in episodic memory . **Psychological Review**, 80, 352-373.

Van Lehn, K. (1983). **Felicity conditions for human skill acquisition: Validating an AI-based theory.** Technical Report CIS-21, Xerox Parc, Palo Alto.

TABLE 1
Dialogue with LISP Tutor

I want you to define a function called ROTATER. It takes 1 argument called arg. In defining this function we'll use (a b c d) as a typical example of arg. This function is supposed to rotate the argument list. For our sample case, the result would be (d a b c).

What is the LISP code to define ROTATER?

Define rotater: **(defun rotater (arg) (cons (last arg) (cdr arg)))**

```
**********************************************
*                    CODE                    *
*                                            *
*    (defun rotater (arg) (cons <1> <2>))    *
*                                            *
**********************************************
```

Your code for <1>: (last arg) produces (d), which is a list and you need an atom.

What LISP code are you going to write to get the last element of the argument list? Your answer will go in place of <1> in the code. You are trying to come up with an atom, like d in our example.

Code for <1>: **d**

Your code for <1>, d, is a good example for the last element of the argument list, but it is not the general code. We need LISP code that will evaluate to d in our example case. <1> is the last item of the argument list. CAR gets the first element of a list, but since this item is the last element, we need to REVERSE the list first and then take the CAR. Taking the CAR of the REVERSE of (a b c d) gives us d.

```
**********************************************
*                    CODE                    *
*                                            *
*    (defun rotater (arg)                    *
*          (cons (car (reverse <3>)) <2>))   *
*                                            *
**********************************************
```

What are you going to write to go in place of <3> in the code? <3> is a list. A typical example of the result is (a b c d).

Code for <3>: **arg**

Good. This makes your code into the following:

```
**********************************************
*                    CODE                    *
*                                            *
*    (defun rotater (arg)                    *
*          (cons (car (reverse arg)) <2>))   *
*                                            *
**********************************************
```

Your previous code for <2>, (cdr arg), evaluates to (b c d), while what you want is (a b c). Remember that CDR only takes an element off of the front of a list.

What LISP code are you going to write to get all but the last of the argument list? Your answer will go in place of <2> in the code. You are trying to come up with a list, like (a b c) in our example.

Code for <2>: **I don't know**

That is OK. I'll help you out. The following screen will display a menu of choices. You have to decide which action you want to take and then press the corresponding number. One option on every menu is to get help about LISP in general.

Remember that you are trying to get all but the last of the argument list. You are replacing <2> in the code. The result is a list, like (a b c) in this case.

```
PRESS:              IF YOU WANT TO:
  1          transform (a b c d) into what we want
  2          solve for separate pieces of (a b c)
  3          have the tutor choose
  4          get information about LISP
```

Menu choice: **3**

OK. I'll tell you the right answer.

We'll try and work with something we know how to code in order to produce the code we want. We'll use (a b c) as our illustrative example for the tail of the result of rotating the argument list and we'll use (a b c d) as our example for the argument list.

```
**********************************************
*                    PLAN                    *
*                                            *
*     TRANSFORM:  (a b c d)  INTO  (a b c)   *
*                                            *
**********************************************
```

What can we do to change (a b c d) into (a b c)?

```
PRESS:              IF YOU WANT TO:
  1          remove d from (a b c d)
  2          get each element and gather them into a list
  3          have the tutor choose
  4          get information about LISP
```

Menu choice: **2**

No. That is not general enough. If you tried your function with a list with more than 3 elements, it would not work.

Menu choice: 1

Sounds good.

```
**********************************************
*                    PLAN                    *
*                                            *
*   REMOVE d FROM  (a b c d)  TO GET (a b c) *
*                                            *
**********************************************
```

What function will remove just 1 element from a list and return the result?

Function name: **last**

If you apply LAST to (a b c d) you will get (d) whereas what you want is (a b c). You want to remove the last element of the argument list not extract it.

Let's try again.

Function name: **cdr**

Good! Your answer so far is correct, but I want to point out one thing. Remember that in this case, d is the last element of the argument list. So, in order to apply CDR, you have to get d in the first position of the list. We will represent this list you have to get by (d ...)

What can we do to change (a b c d) into (d ...)?

```
PRESS:              IF YOU WANT TO:
  1           get d to the front of the list
  2           go down the list until you get d
  3           have the tutor choose
  4           get information about LISP
```

Menu choice: 1

That answer is correct.

```
**************************************************
*                      PLAN                      *
*                                                *
*     TRANSFORM:  (a b c d)  INTO  (d ...)       *
*                                                *
**************************************************
```

What function will get d to the front of the list?

Function name: **reverse**

That is a good answer. But, remember that you are trying to get (a b c). The plan so far for getting the argument list minus the last element is the following: (cdr (reverse '(a b c d))) What does this evaluate to in LISP?

Result: **(c b a)**

Very good.

```
**************************************************
*                  EXAMPLE CODE                  *
*                                                *
*   (cdr (reverse '(a b c d)))  =  (c b a)       *
*                                                *
**************************************************
```

What can we do to change (c b a) into (a b c)?

```
PRESS:              IF YOU WANT TO:
  1           successively get all elements of (c b a)
  2           combine (c) with (a b)
  3           flip over (c b a)
  4           have the tutor choose
  5           get information about LISP
```

Menu choice: 1

No. We don't have any way of repeatedly getting elements. If we did have a procedure to take off and replace exactly 3 items, it wouldn't work on other size lists.

We'll give you another shot.

Menu choice: 3

That is a correct answer. Very good.

```
**********************************************
*                  PLAN                      *
*                                            *
*       TRANSFORM:  (c b a)  INTO  (a b c)   *
*                                            *
**********************************************
```

What function will flip over (c b a)?

Function name: **don't have the faintest**

You can get (a b c) by flipping over (c b a) with the function REVERSE.

Current Example Code ---> (reverse (cdr (reverse '(a b c d)))) = (a b c)

Now that you've written code to achieve (a b c), what is the general LISP code to replace the code for our example? Your answer will go in place of <2> in the code.

Code for <2>: **(reverse (cdr (reverse '(a b c d))))**

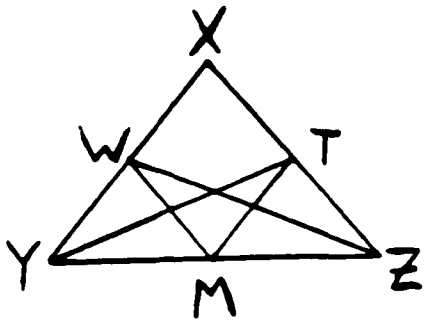
That is the code using the example. I want the GENERAL LISP code, using the variables in our function definition.

Let's attempt it again

Code for <2>: **(reverse (cdr (reverse arg)))**

!!! YOU ARE DONE !!!

```
**********************************************
*              FINAL CODE                    *
*                                            *
*      (defun rotater (arg)                  *
*         (cons (car (reverse arg))          *
*            (reverse                        *
*               (cdr                         *
*                 (reverse arg)))))          *
*                                            *
**********************************************
```



GIVEN: $\overline{XY} \cong \overline{XZ}$, $\angle WMY \cong \angle TMZ$
M midpoint of $\overline{YZ}$

PROVE: $\overline{YT} \cong \overline{ZW}$

STATEMENT	REASON
M is midpoint of $\overline{YZ}$	Given
$\overline{YM} \cong \overline{MZ}$	Definition of midpoint
$XY \cong XZ$	Given
$\angle XYZ \cong \angle XZY$	base angles of isoceles triangles
$\angle WMY \cong \angle TMZ$	Given
$\triangle WMY \cong \triangle TMZ$	Angle-side-angle (ASA)
$\overline{WY} \cong \overline{TZ}$	Corresponding parts
$\triangle WYZ \cong \triangle TZY$	side-angle-side (SAS)
$\overline{YT} \cong \overline{ZW}$	Corresponding parts

Fig. 1

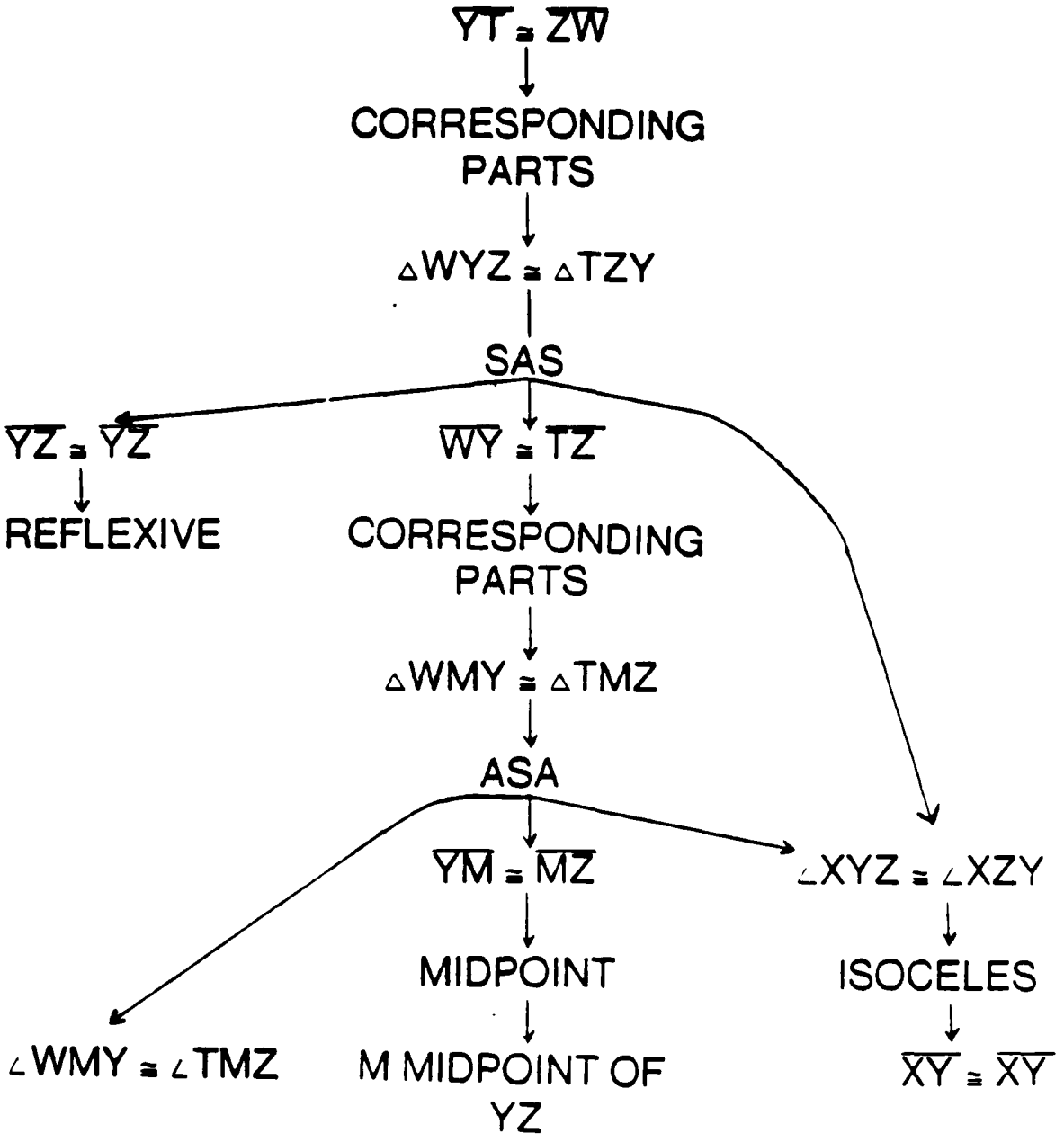

$\overline{YT} \cong \overline{ZW}$
CORRESPONDING PARTS
$\triangle WYZ \cong \triangle TZY$
SAS
$\overline{YZ} \cong \overline{YZ}$
REFLEXIVE
$\overline{WY} \cong \overline{TZ}$
CORRESPONDING PARTS
$\triangle WMY \cong \triangle TMZ$
ASA
$\overline{YM} \cong \overline{MZ}$
MIDPOINT
M MIDPOINT OF YZ
$\angle XYZ \cong \angle XZY$
ISOCELES
$\overline{XY} \cong \overline{XY}$
$\angle WMY \cong \angle TMZ$

Fig 2.

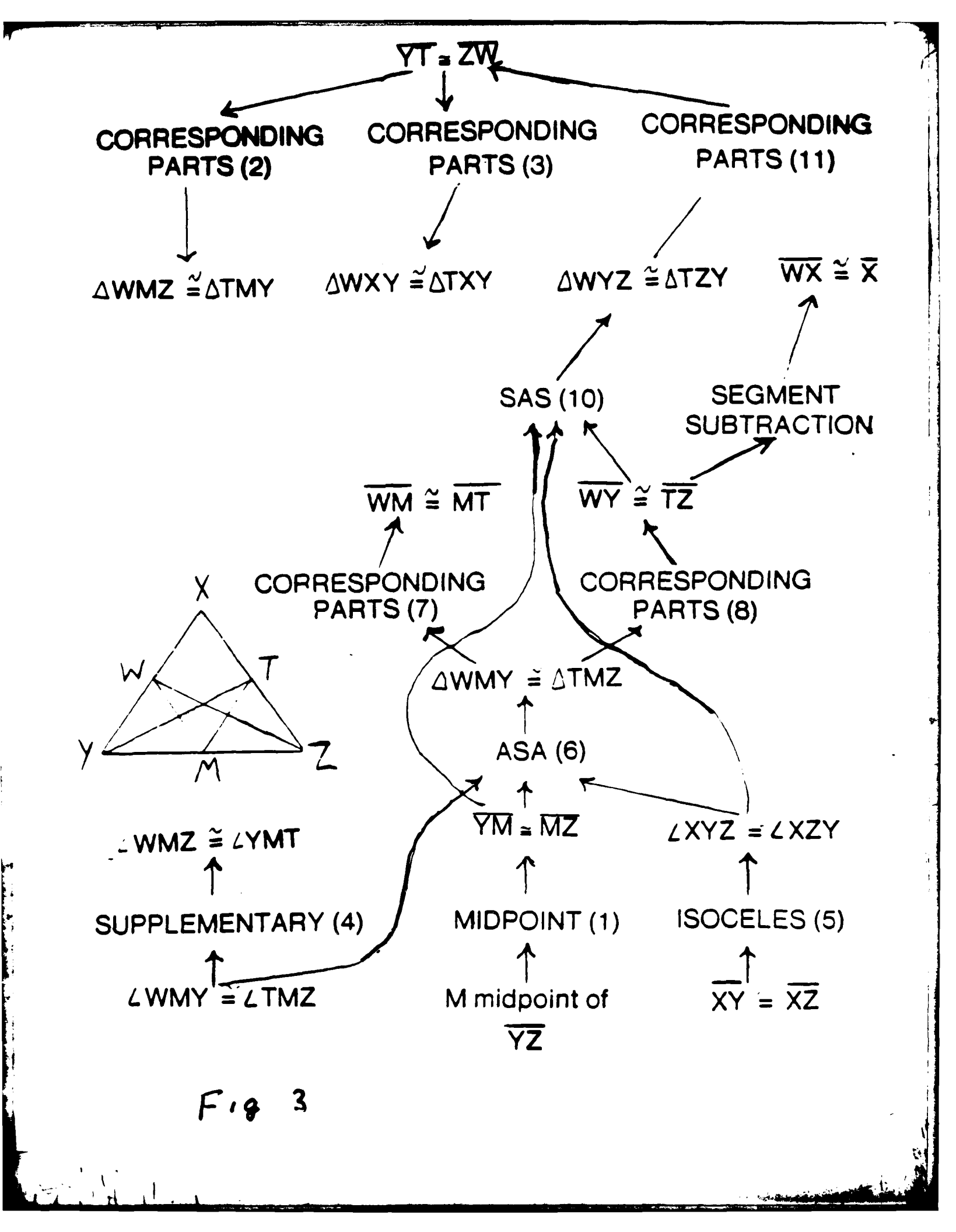
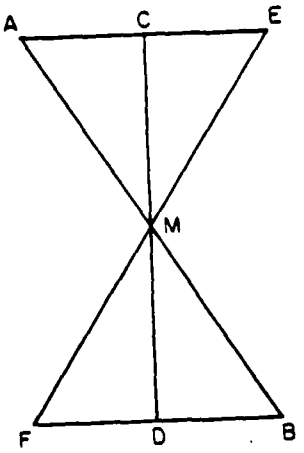

YT ≅ ZW
CORRESPONDING PARTS (2)
CORRESPONDING PARTS (3)
CORRESPONDING PARTS (11)
ΔWMZ ≅ ΔTMY
ΔWXY ≅ ΔTXY
ΔWYZ ≅ ΔTZY
WX ≅ XZ
SAS (10)
SEGMENT SUBTRACTION
WM ≅ MT
WY ≅ TZ
CORRESPONDING PARTS (7)
CORRESPONDING PARTS (8)
X
W
T
Y
M
Z
ΔWMY ≅ ΔTMZ
ASA (6)
∠WMZ ≅ ∠YMT
YM ≅ MZ
∠XYZ ≅ ∠XZY
SUPPLEMENTARY (4)
MIDPOINT (1)
ISOCELES (5)
∠WMY ≅ ∠TMZ
M midpoint of YZ
XY ≅ XZ

Fig 3



GIVEN: M is the midpoint of $\overline{AB}$ and $\overline{CD}$

PROVE: M is the midpoint of $\overline{EF}$

Fig 4

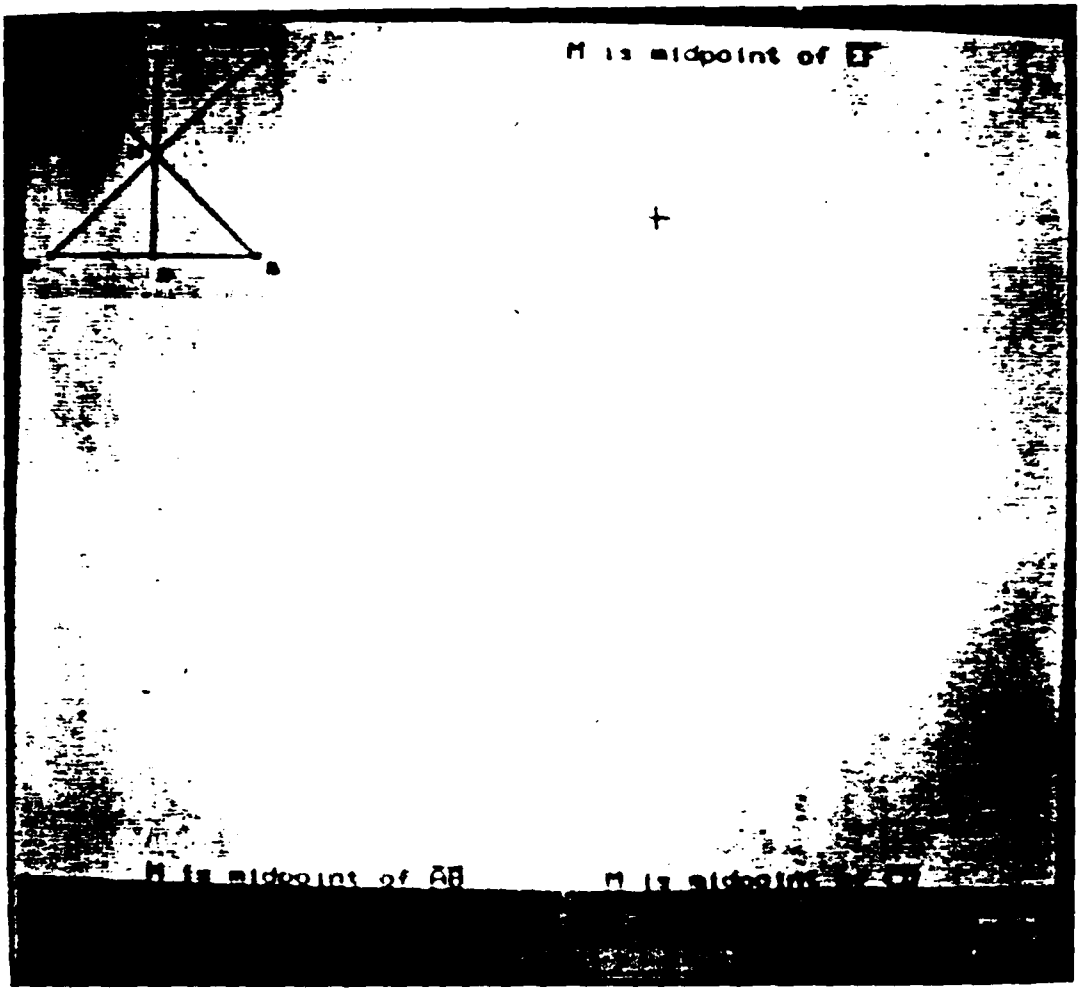


Figure 5a

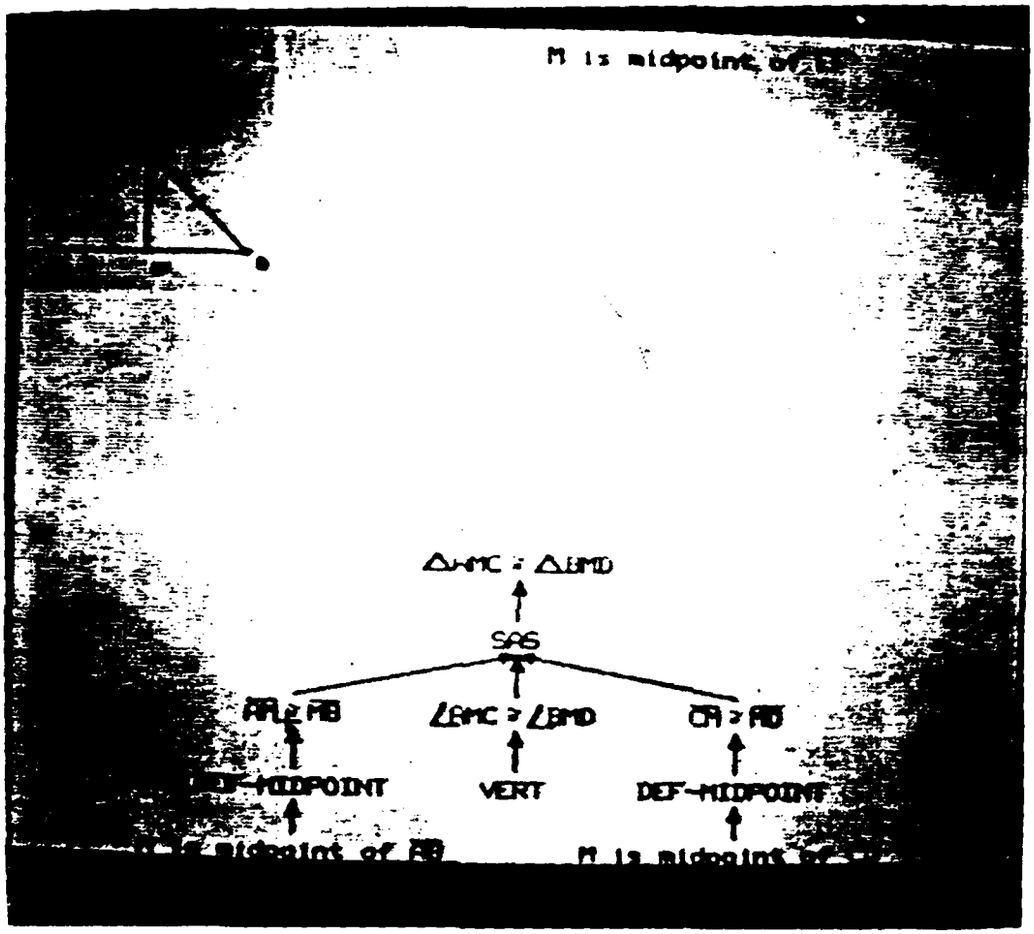

M is midpoint of
△AMC ≅ △BMD
SAS
∠AMC ≅ ∠BMD
VERT
DEF-MIDPOINT
M is midpoint of

Figure 5b

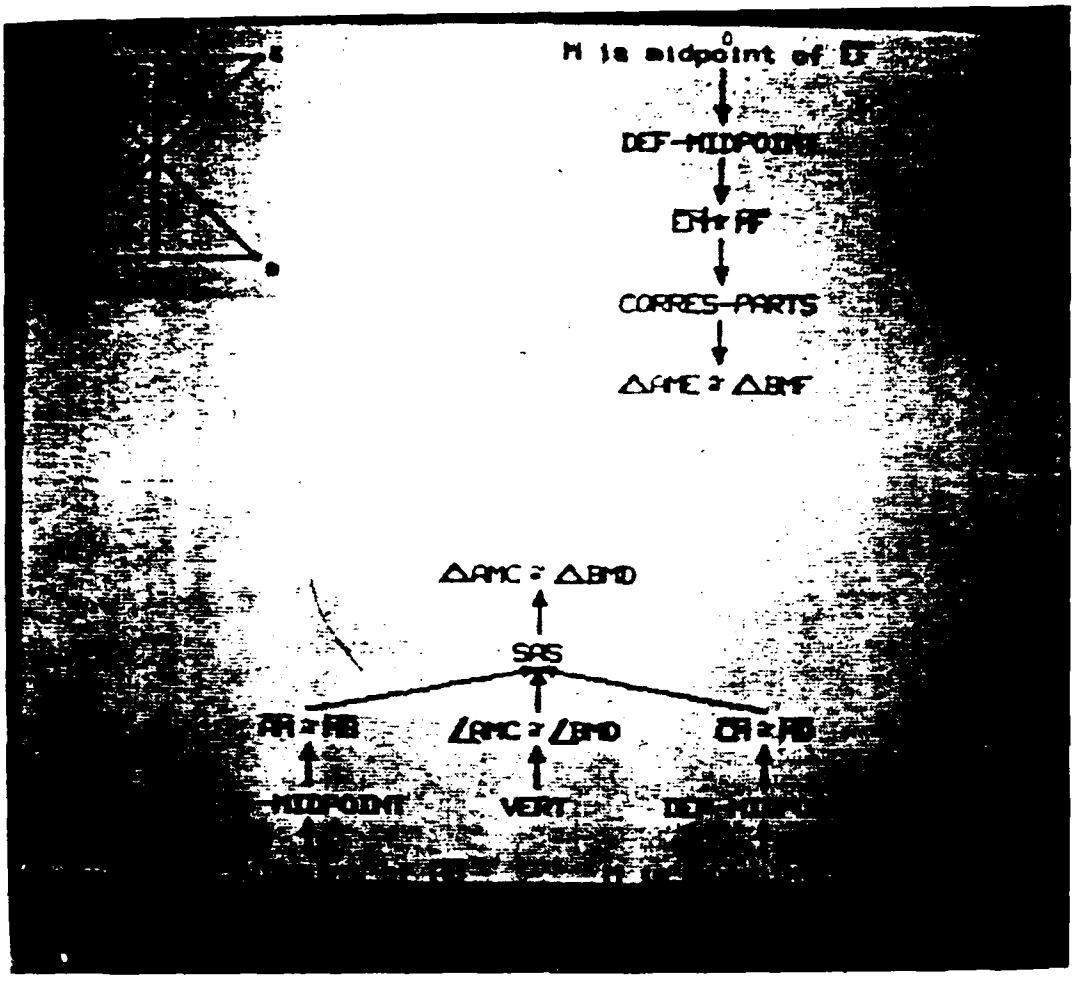


Figure 5c

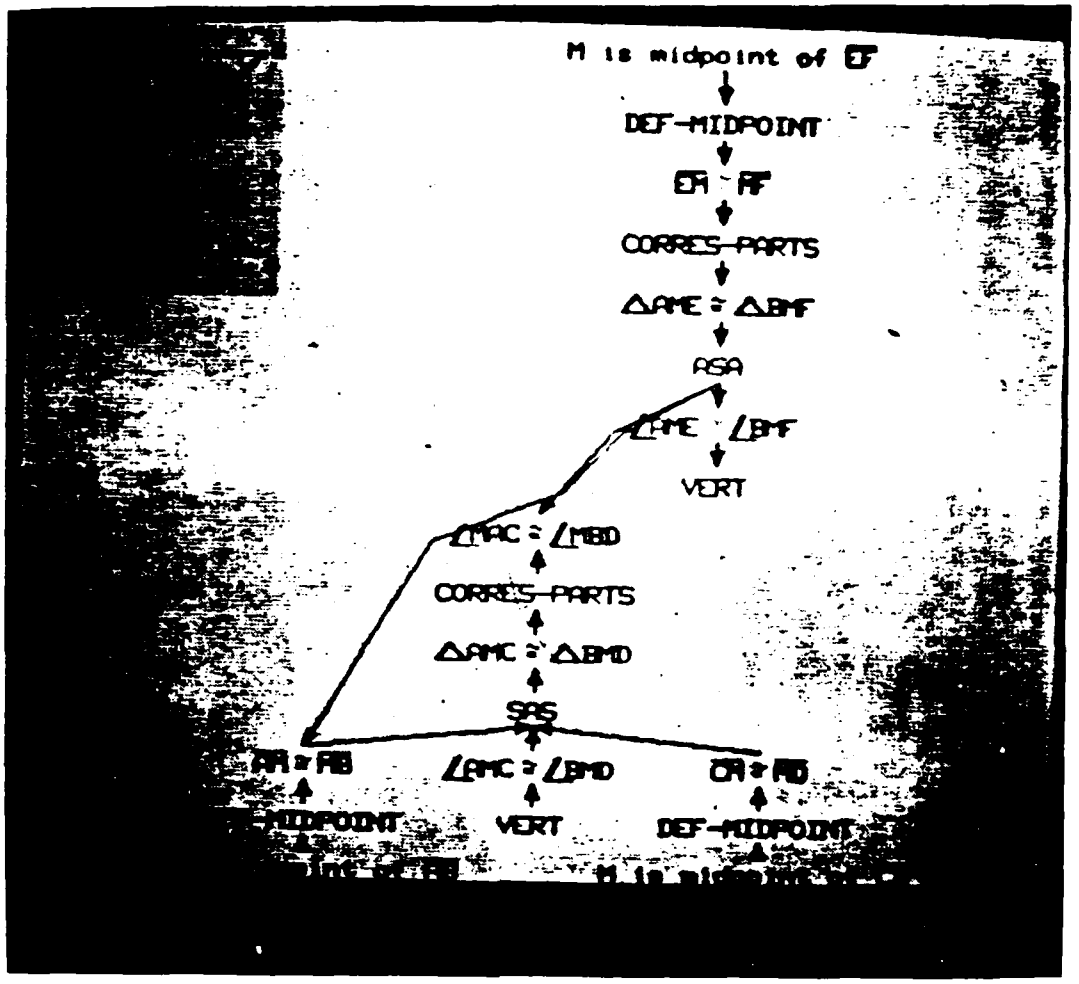
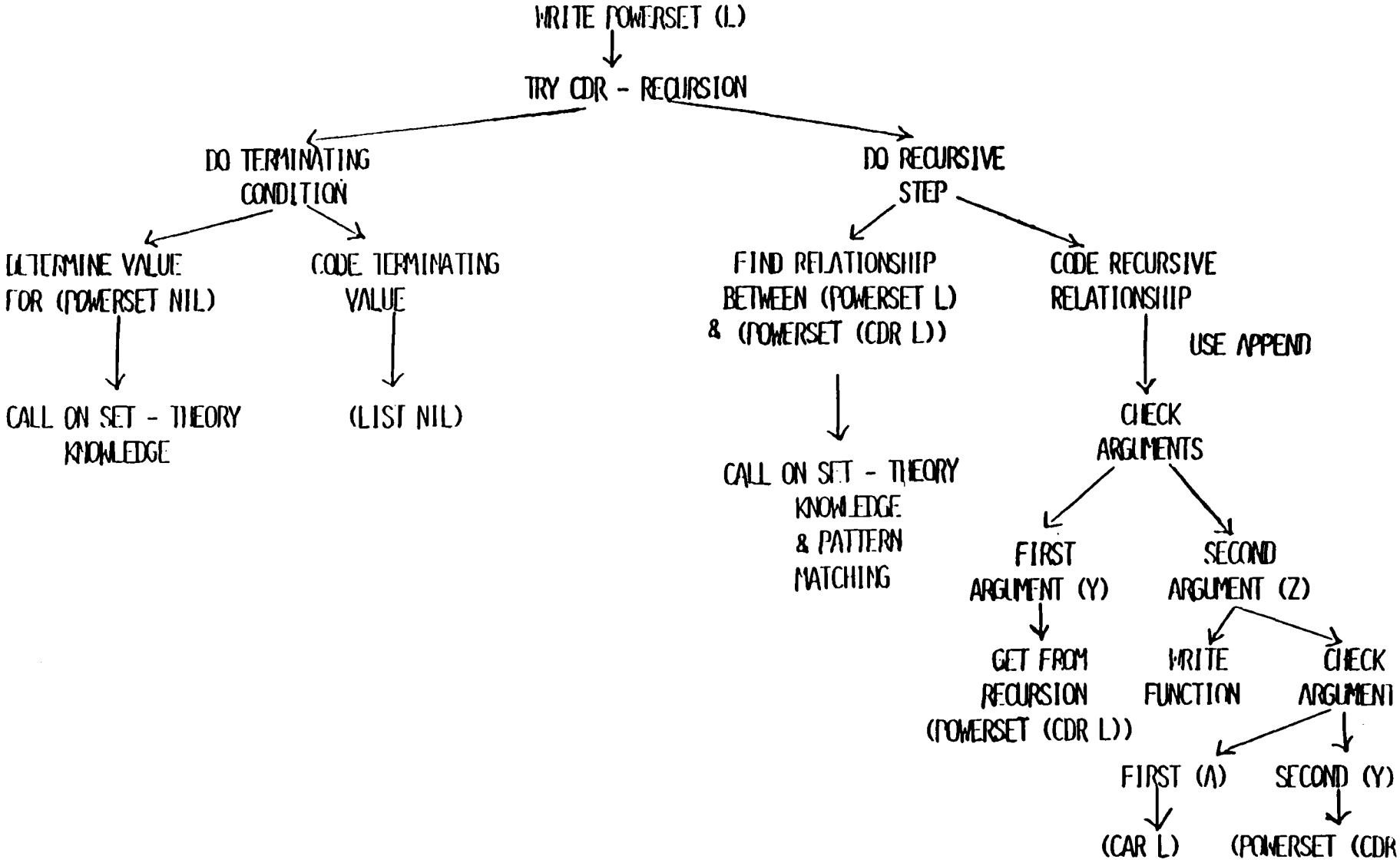

M is midpoint of EF
DEF-MIDPOINT
EM = MF
CORRES-PARTS
ΔAME ≅ ΔBMF
ASA
∠AME ≅ ∠BMF
VERT
∠MAC ≅ ∠MBD
CORRES-PARTS
ΔAMC ≅ ΔBMD
SAS
AM = MB
∠AMC ≅ ∠BMD
CM = MD
MIDPOINT
VERT
DEF-MIDPOINT

Figure 5d


WRITE POWERSET (L)
TRY CDR - RECURSION
DO TERMINATING CONDITION
DO RECURSIVE STEP
DETERMINE VALUE FOR (POWERSET NIL)
CODE TERMINATING VALUE
FIND RELATIONSHIP BETWEEN (POWERSET L) & (POWERSET (CDR L))
CODE RECURSIVE RELATIONSHIP
USE APPEND
CALL ON SET - THEORY KNOWLEDGE
(LIST NIL)
CALL ON SET - THEORY KNOWLEDGE & PATTERN MATCHING
CHECK ARGUMENTS
FIRST ARGUMENT (Y)
SECOND ARGUMENT (Z)
GET FROM RECURSION (POWERSET (CDR L))
WRITE FUNCTION
CHECK ARGUMENT
FIRST (A)
SECOND (Y)
(CAR L)
(POWERSET (CDR

What code are you going to write for <1>. This should be the argument list. It would be the list, (a b c d), in the example.

>>>lis

Code:

```
(defun rotater (lis)
    (append (last <1>) <2>)
```

Example: (rotater '(a b c d)) = (d a b c)

You are designing an algorithm for <2>

You want to get a list of all but the last item of the argument list. How are you going to do this?

1. Get all elements and gather them into a list.
2. Remove D from (A B C D)
3. Let the tutor choose
4. Get information about LISP

>>>2

Algorithm:
INITIAL: (A B C D)
GOAL: (A B C D) --> (A B C)

Code:

```
(defun rotater (lis)
    (append (last lis) <2>)
```

Example: (rotater '(a b c d)) = (d a b c)

Navy

1 Robert Ahlers
Code N711
Human Factors Laboratory
NAVTRAEQUIPCEN
Orlando, FL 32813

1 Dr. Meryl S. Baker
Navy Personnel R&D Center
San Diego, CA 92152

1 Code N711
Attn: Arthur S. Blaiwes
Naval Training Equipment Center
Orlando, FL 32813

1 Dr. Richard Cantone
Navy Research Laboratory
Code 7510
Washington, DC 20375

1 Dr. Susan Chipman
Code 442PT
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217

1 Dr. Stanley Collyer
Office of Naval Technology
800 N. Quincy Street
Arlington, VA 22217

1 CDR Mike Curran
Office of Naval Research
800 N. Quincy St.
Code 270
Arlington, VA 22217

1 DR. PAT FEDERICO
Code P13
NPRDC
San Diego, CA 92152

1 Dr. Jude Franklin
Code 7510
Navy Research Laboratory
Washington, DC 20375

1 LT Steven D. Harris, MSC, USN
RFD 1, Box 243
Riner, VA 24149

Navy

1 Dr. Jim Hollan
Code 14
Navy Personnel R & D Center
San Diego, CA 92152

1 Dr. Ed Hutchins
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Norman J. Kerr
Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, TN 38054

1 Dr. Peter Kincaid
Training Analysis & Evaluation Group
Dept. of the Navy
Orlando, FL 32813

1 Dr. William L. Maloy (02)
Chief of Naval Education and Training
Naval Air Station
Pensacola, FL 32508

1 Dr. Joe McLachlan
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr William Montague
NPRDC Code 13
San Diego, CA 92152

1 Technical Director
Navy Personnel R&D Center
San Diego, CA 92152

6 Commanding Officer
Naval Research Laboratory
Code 2627
Washington, DC 20390

1 Office of Naval Research
Code 433
800 N. Quincy SStreet
Arlington, VA 22217

6 Personnel & Training Research Group
Code 442PT
Office of Naval Research
Arlington, VA 22217

Navy

1 Office of the Chief of Naval Operations
Research Development & Studies Branch
OP 115
Washington, DC 20350

1 LT Frank C. Petho, MSC, USN (Ph.D)
CNET (N-432)
NAS
Pensacola, FL 32508

1 Dr. Gil Ricard
Code N711
NTEC
Orlando, FL 32813

1 Dr. Robert G. Smith
Office of Chief of Naval Operations
OP-987H
Washington, DC 20350

1 Dr. Alfred F. Smode, Director
Department N-7
Naval Training Equipment Center
Orlando, FL 32813

1 Dr. Richard Snow
Liaison Scientist
Office of Naval Research
Branch Office, London
Box 39
FPO New York, NY 09510

1 Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Frederick Steinheiser
CNO - OP115
Navy Annex
Arlington, VA 20370

1 Roger Weissinger-Baylon
Department of Administrative Sciences
Naval Postgraduate School
Monterey, CA 93940

1 Mr John H. Wolfe
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Wallace Wulfeck, III
Navy Personnel R&D Center
San Diego, CA 92152

Marine Corps

1 H. William Greenup
Education Advisor (E031)
Education Center, MCDEC
Quantico, VA 22134

1 Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217

1 DR. A.L. SLAFKOSKY
SCIENTIFIC ADVISOR (CODE RD-1)
HQ, U.S. MARINE CORPS
WASHINGTON, DC 20380

Army

1 Technical Director
U. S. Army Research Institute for the
Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Mr. James Baker
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Milton S. Katz
Training Technical Area
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Marshall Narva
US Army Research Institute for the
Behavioral & Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Harold F. O'Neil, Jr.
Director, Training Research Lab
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Commander, U.S. Army Research Institute
for the Behavioral & Social Sciences
ATTN: PERI-BR (Dr. Judith Orasanu)
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Joseph Psotka, Ph.D.
ATTN: PERI-IC
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

1 Dr. Robert Sasmor
U. S. Army Research Institute for the
Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Air Force

1 U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, DC 20332

1 Dr. Earl A. Alluisi
HQ, AFHRL (AFSC)
Brooks AFB, TX 78235

1 Mr. Raymond E. Christal
AFHRL/MOE
Brooks AFB, TX 78235

1 Bryan Dallman
AFHRL/LRT
Lowry AFB, CO 80230

1 Dr. Alfred R. Fregly
AFOSR/NL
Bolling AFB, DC 20332

1 Dr. Genevieve Haddad
Program Manager
Life Sciences Directorate
AFOSR
Bolling AFB, DC 20332

1 Dr. T. M. Longridge
AFHRL/OTE
Williams AFB, AZ 85224

1 Dr. John Tangney
AFOSR/NL
Bolling AFB, DC 20332

1 Dr. Joseph Yasatuke
AFHRL/LRT
Lowry AFB, CO 80230

Department of Defense

12 Defense Technical Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC

1 Military Assistant for Training and
Personnel Technology
Office of the Under Secretary of Defens
for Research & Engineering
Room 3D129, The Pentagon
Washington, DC 20301

1 Major Jack Thorpe
DARPA
1400 Wilson Blvd.
Arlington, VA 22209

1 Dr. Robert A. Wisher
OUSDRE (ELS)
The Pentagon, Room 3D129
Washington, DC 20301

Civilian Agencies

1 Dr. Patricia A. Butler
NIE-BRN Bldg, Stop # 7
1200 19th St., NW
Washington, DC 20208

1 Edward Esty
Department of Education, OERI
MS 40
1200 19th St., NW
Washington, DC 20208

1 Dr. Arthur Melmed
724 Brown
U. S. Dept. of Education
Washington, DC 20208

1 Dr. Andrew R. Molnar
Office of Scientific and Engineering
Personnel and Education
National Science Foundation
Washington, DC 20550

1 Dr. Frank Withrow
U. S. Office of Education
400 Maryland Ave. SW
Washington, DC 20202

1 Dr. Joseph L. Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

Private Sector

1 Mr. Avron Barr
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. John Black
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 Dr. John S. Brown
XEROX Palo Alto Research Center
3333 Coyote Road
Palo Alto, CA 94304

1 Dr. Glenn Bryan
6208 Poe Road
Bethesda, MD 20817

1 Dr. Bruce Buchanan
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. Jaime Carbonell
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

1 Dr. Pat Carpenter
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. Micheline Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. William Clancey
Department of Computer Science
Stanford University
Stanford, CA 94306

1 Dr. Michael Cole
University of California
at San Diego
Laboratory of Comparative
Human Cognition - D003A
La Jolla, CA 92093

Private Sector

1 Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Lynn A. Cooper
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. Thomas M. Duffy
Department of English
Carnegie-Mellon University
Schenley Park
Pittsburgh, CA 15213

1 Dr. Anders Ericsson
Department of Psychology
University of Colorado
Boulder, CC 80309

1 Dr. Paul Feltovich
Department of Medical Education
Southern Illinois University
School of Medicine
P.O. Box 3926
Springfield, IL 62708

1 Mr. Wallace Feurzeig
Department of Educational Technology
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02238

1 Dr. Dexter Fletcher
University of Oregon
Department of Computer Science
Eugene, OR 97403

1 Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

1 Dr. Michael Genesereth
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. Dedre Gentner
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02138

Private Sector

1 Dr. Don Gentner
Center for Human Information Processing
University of California, San Diego
La Jolla, CA 92093

1 Dr. Robert Glaser
Learning Research & Development Center
University of Pittsburgh
3939 O'Hara Street
PITTSBURGH, PA 15260

1 Dr. Joseph Goguen
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

1 Dr. Daniel Gopher
Faculty of Industrial Engineering
& Management
TECHNION
Haifa 32000
ISRAEL

1 Dr. Bert Green
Johns Hopkins University
Department of Psychology
Charles & 34th Street
Baltimore, MD 21218

1 DR. JAMES G. GREENO
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

1 Dr. Barbara Hayes-Roth
Department of Computer Science
Stanford University
Stanford, CA 95305

1 Dr. Frederick Hayes-Roth
Teknowledge
525 University Ave.
Palo Alto, CA 94301

1 Dr. Earl Hunt
Dept. of Psychology
University of Washington
Seattle, WA 98105

1 Dr. Marcel Just
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Private Sector

1 Dr. Scott Kelso
Haskins Laboratories, Inc
270 Crown Street
New Haven, CT 06510

1 Dr. David Kieras
Department of Psychology
University of Arizona
Tuscon, AZ 85721

1 Dr. Walter Kintsch
Department of Psychology
University of Colorado
Boulder, CO 80302

1 Dr. Stephen Kosslyn
1236 William James Hall
33 Kirkland St.
Cambridge, MA 02138

1 Dr. Pat Langley
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. Jill Larkin
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Alan Lesgold
Learning R&D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

1 Dr. Jim Levin
University of California
at San Diego
Laboratory for Comparative
Human Cognition - D003A
La Jolla, CA 92093

1 Dr. Michael Levine
Department of Educational Psychology
210 Education Bldg.
University of Illinois
Champaign, IL 61801

1 Dr. Marcia C. Linn
Lawrence Hall of Science
University of California
Berkeley, CA 94720

Private Sector

1 Dr. Jay McClelland
Department of Psychology
MIT
Cambridge, MA 02139

1 Dr. James R. Miller
Computer*Thought Corporation
1721 West Plano Highway
Plano, TX 75075

1 Dr. Mark Miller
Computer*Thought Corporation
1721 West Plano Parkway
Plano, TX 75075

1 Dr. Tom Moran
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

1 Dr. Allen Munro
Behavioral Technology Laboratories
1845 Elena Ave., Fourth Floor
Redondo Beach, CA 90277

1 Dr. Donald A Norman
Cognitive Science, C-015
Univ. of California, San Diego
La Jolla, CA 92093

1 Dr. Jesse Orlansky
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311

1 Prof. Seymour Papert
20C-109
Massachusetts Institute of Technology
Cambridge, MA 02139

1 Dr. Nancy Pennington
University of Chicago
Graduate School of Business
1101 E. 58th St.
Chicago, IL 60637

1 DR. PETER POLSON
DEPT. OF PSYCHOLOGY
UNIVERSITY OF COLORADO
BOULDER, CO 80309

Private Sector

1 Dr. Fred Reif
Physics Department
University of California
Berkeley, CA 94720

1 Dr. Lauren Resnick
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 1521

1 Mary S. Riley
Program in Cognitive Science
Center for Human Information Processing
University of California, San Diego
La Jolla, CA 92093

1 Dr. Andrew M. Rose
American Institutes for Research
1055 Thomas Jefferson St. NW
Washington, DC 20007

1 Dr. Ernst Z. Rothkopf
Bell Laboratories
Murray Hill, NJ 07974

1 Dr. William B. Rouse
Georgia Institute of Technology
School of Industrial & Systems
Engineering
Atlanta, GA 30332

1 Dr. David Rumelhart
Center for Human Information Processing
Univ. of California, San Diego
La Jolla, CA 92093

1 Dr. Michael J. Samet
Perceptronics, Inc
6271 Variel Avenue
Woodland Hills, CA 91364

1 Dr. Roger Schank
Yale University
Department of Computer Science
P.O. Box 2158
New Haven, CT 06520

1 Dr. Walter Schneider
Psychology Department
603 E. Daniel
Champaign, IL 61820

Private Sector

1 Dr. Alan Schoenfeld
Mathematics and Education
The University of Rochester
Rochester, NY 14627

1 Mr. Colin Sheppard
Applied Psychology Unit
Admiralty Marine Technology Est.
Teddington, Middlesex
United Kingdom

1 Dr. H. Wallace Sinaiko
Program Director
Manpower Research and Advisory Services
Smithsonian Institution
801 North Pitt Street
Alexandria, VA 22314

1 Dr. Edward E. Smith
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Elliott Soloway
Yale University
Department of Computer Science
P.O. Box 2158
New Haven, CT 06520

1 Dr. Kathryn T. Spoehr
Psychology Department
Brown University
Providence, RI 02912

1 Dr. Robert Sternberg
Dept. of Psychology
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 Dr. Albert Stevens
Bolt Beranek & Newman, Inc.
10 Moulton St.
Cambridge, MA 02238

1 DR. PATRICK SUPPES
INSTITUTE FOR MATHEMATICAL STUDIES IN
THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CA 94305

Private Sector

1 Dr. Kikumi Tatsuoka
Computer Based Education Research Lab
252 Engineering Research Laboratory
Urbana, IL 61801

1 Dr. Maurice Tatsuoka
220 Education Bldg
1310 S. Sixth St.
Champaign, IL 61820

1 Dr. Perry W. Thorndyke
Perceptronics, Inc.
545 Middlefield Road, Suite 140
Menlo Park, CA 94025

1 Dr. Douglas Towne
Univ. of So. California
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

1 Dr. Kurt Van Lehn
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

1 Dr. Keith T. Wescourt
Perceptronics, Inc.
545 Middlefield Road, Suite 140
Menlo Park, CA 94025

1 Dr. Thomas Wickens
Department of Psychology
Franz Hall
University of California
405 Hilgarde Avenue
Los Angeles, CA 90024

1 Dr. Mike Williams
IntelliGenetics
124 University Avenue
Palo Alto, CA 94301


DATE
ILME